Integration

Steve Rotenberg
CSE291: Physics Simulation
UCSD
Spring 2019

Integration

- Physics simulation in the Newtonian realm involves working with forces
- As we've seen, forces relate to accelerations through Newton's Second Law $f=ma$
- Ultimately however, we need to compute positions in order to advance the simulation forward and visualize what's happening
- Position is the integral of velocity and velocity is the integral of acceleration
- Therefore, the process of integration will be central to physics simulation

Analytical (Symbolic) Integration

- If we have a relatively simple mathematical function, we can usually compute an analytical integral
- For example, if our function is a polynomial like:

$$f(t) = 3t^2 + 4t + 5$$

- then we can compute the integral as

$$\int f(t)dt = t^3 + 2t^2 + 5t + c$$

- Analytical integration calculates an exact solution to the integral.

- NOTE: The usage of the word *analytical* is consistent with its usage in this field, and is often used in discussions of *analytical* vs. *numerical* methods. Some people prefer the term *symbolic*, as *analytical* has a different meaning in mathematics, relating to *analytic functions*.

Numerical Integration

- However, many mathematical functions can’t be integrated analytically, and this applies to most of the situations we’ll be interested in.
- We will also often have to deal with forces that are generated by processes other than mathematical functions. For example, we might have an interactive vehicle simulation where the throttle setting is controlled by a human operator. We clearly won’t find a mathematical equation that defines the throttle as a function of time.
- Therefore, we will accept that we will rely on *numerical integration* techniques throughout the entire quarter.
- Numerical integration uses iteration and approximation to compute “brute force” results, and so can suffer from problems with accuracy and stability.
- Still, we have little choice if we want to simulate complex problems, so we must understand these properties in order to make use of numerical integration techniques.

Numerical Dynamics Simulation

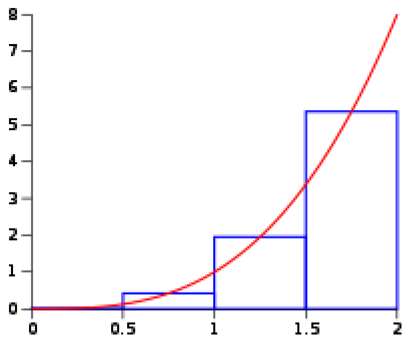
- In general, the process of numerical dynamics simulation can be summarized as:

```
Specify initial conditions for time $t_0$
While (not finished) {
    • Evaluate all forces in current configuration at time $t_n$ (and possibly other
      nearby times) and use these to compute all accelerations
    • Integrate accelerations over some finite time step $\Delta t$ to advance everything
      to new positions (and velocities) at new time $t_{n+1}$
    • Display/store/analyze results
}
```

- NOTE: Technically, this refers to an *explicit* integration method vs. an *implicit* one which looks similar. More on this later.

Area Under a Curve

- Remember that the integral of a function is equal to the area under the curve defined by the function
- One way to approximate the integral is to slice it up into a bunch of rectangles (or trapezoids) and add up their areas
- This works well for explicit functions where we can evaluate the function at any point
- We can apply a similar idea to differential equations (with some modifications)



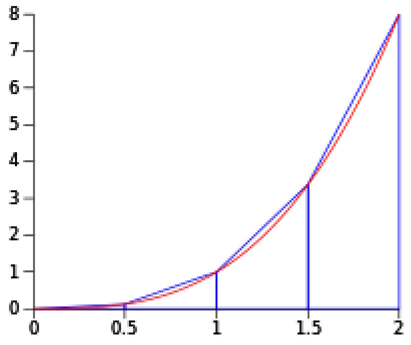


Image source: Wikipedia

Ordinary Differential Equations

- A *differential equation* is an equation that relates a function with its derivatives
- An *ordinary differential equation* (ODE) is a differential equation with one or more functions of **one** independent variable (and the derivatives of those functions)
- For example, the motion of a 3D particle can be described as 3 separate equations of one variable, and so qualifies as an ODE

Partial Differential Equations

- A *partial differential equation* (PDE) is a differential equation that contains functions of multiple variables and their partial derivatives
- These show up in field equations such as in fluid dynamics
- We will get to these later in the quarter
- Many of the integration methods we will look at today apply to both ODEs and PDEs, but we will mainly stick to discussing ODEs today

Initial Value Problems

- An *initial value problem* is a differential equation where the value of the unknown function is specified at some point in the domain
- In physics, we would usually be referring to the time domain, and the value we will specify would usually be at the initial time
- The dynamics equations we will look at throughout the quarter will [almost] always be initial value problems

Initial Value Problems

- The general approach to evaluating initial value problems is similar to the way we calculate the area under a curve
- We start an initial value specified at the initial time
- We then evaluate the derivatives at the current time, and then advance forward by some time step and approximate the value at the new time
- This repeats until we decide we’re done
- Each time step is effectively adding one more rectangle to compute the area under a curve
- By using smaller time steps, we can expect to achieve better accuracy

Stability

- There are several potential problems we need to know about
- One of those is the issue of stability
- If our time step is too large, this can lead to big accuracy problems or even lead to the simulation ‘blowing up’
- For example, if we’re trying to simulate a very high frequency oscillation (like a tuning fork vibrating) with time steps larger than oscillation period itself, we will run into big problems
- In fact, the time step should be significantly smaller than the period to ensure stability

Stiffness

- A *stiff* problem refers to a problem that requires very small time steps to achieve stability
- It's actually a fairly vague term without a precise definition, but it generally means that we need to look very seriously at the integration process and possibly resort to specialized methods
- Very small time steps will not only lead to slow simulations, but can actually get to the point where floating point roundoff will start to dominate and cause its own share of problems
- In physics, stiff problems show up when we have very stiff materials, such as springs with very high spring constants. It is often easier to simulate soft spongy materials compared to stiff ones
- We will look at some integration techniques that are designed to address stiff problems

Accuracy

- Another issue that we need to consider when choosing an integration scheme is accuracy
- In general, using smaller time steps will improve the accuracy of any integration scheme (up to the point where roundoff errors start to dominate)
- However, some schemes are inherently more or less accurate than others, and accuracy usually comes at a price of additional computation

Approximation Order

- The *local truncation error* of an integration scheme is the approximation error made in a single step
- It is the difference between the numerical solution after one step from the exact solution
- It is common to compare an integration scheme to the full Taylor series expansion of the solution
- Approximation schemes will be the same (or similar) to a truncated Taylor series (i.e., only the first few terms of the Taylor series)
- The truncated part represents the error, which will be a function of Δt^n where n is the first truncated term

Approximation Order

- For example, consider the Taylor expansion of the function $y(t)$ near t_0:

$$y(t_0 + \Delta t) = y(t_0) + y'(t_0)\Delta t + \frac{1}{2}y''(t_0)\Delta t^2 + O(\Delta t^3)$$

- Where $O(\Delta t^3)$ refers to additional terms based on Δt^3 or higher powers
- The forward Euler method we looked at briefly in the last lecture approximates this with just:

$$y(t_0 + \Delta t) = y(t_0) + y'(t_0)\Delta t$$

- In other words, we are dropping all $O(\Delta t^2)$ terms or higher, which implies the error is on the order of Δt^2
- Thus, the forward Euler method is known as a first order method, in that it is accurate up to $O(\Delta t)$
- Higher order methods will not always be more accurate in all cases, but they should have much better convergence towards an accuracy as Δt gets smaller

Time Steps

- We have seen that the choice of time step can have significant implications on:
  - Stability
  - Accuracy
  - Performance
- Usually, one has some upper limit on the time step. For example, if we ultimately want to produce a 30 frame per second animation, we would have 1/30 sec as our upper limit
- Stability and accuracy however, might require a smaller time step
- In some cases, one can simply choose a time step that works and stick with that
- In other cases, it is best to allow the system to change the time step automatically in order to adapt to the change. We will look at adaptive time step schemes later in the lecture

CFL Condition

- In a lot of situations, one can actually calculate what the time step should be in order to achieve accuracy and stability
- The ideal time step will change as the simulation proceeds, and needs to be re-evaluated each step of the simulation
- For example, in a mass-spring simulation, one can base some of these calculations on the ratio of the spring stiffness constants to the masses of the particles
- In field equations (like fluid dynamics), one can base these on the Courant-Friedrichs-Lewy (CFL) condition
- As field equations are typically solved on some sort of grid, the condition ensures that the time step is chosen so as to prevent anything from moving more than the distance of the grid size in a single time step
- We will examine the CFL condition in more detail when we talk about fluids and field equations

Real Time Stepping

- When doing an interactive simulation, it is tempting to use the real time clock to determine the time step
- This is usually a bad idea unless certain precautions are taken
- One big problem is that the operating system might need to do some work that causes a brief pause in the simulation
- For example, in most systems, dragging the window around with the mouse will cause the program to pause. When the drag is finished, the simulation gets a single frame with a very large time step, potentially causing major instability
- For this reason, you definitely want to put some sort of upper limit on the time step
- Another option is to run multiple small steps to add up to the total step (known as *oversampling*). For example, if your simulation needs 1/60 second steps to run stable and the system clock says that 1/15 of a second has past since the last frame, then you would simulate 4 frames at 1/60
- You would want to put an upper limit on the number of frames to prevent runaway problems where each frame is slower than the last

Integration Methods

Initial Value Problems

- Let's say we have a function $y(t)$ that we need to integrate
- It's derivative $y'(t)$ is a function of time and of the current value of y:

$$y'(t) = f(t, y)$$

- We have some initial value for y at time t_0

$$y(t_0) = y_0$$

- We have some time step Δt that we want to use to advance the simulation forward in time:

$$t_{i+1} = t_i + \Delta t$$

- And we want to evaluate y_i at each time t_i

Forward Euler Integration

- Perhaps the simplest method for integrating is the forward (or explicit) Euler method:

$$y_{i+1} = y_i + f(t_i, y_i)\Delta t$$

- It evaluates the derivative function f at the current time t_i and uses that to advance the entire time step forward
- It is easy to implement but can suffer from problems with both accuracy and stability

Double Integration

- For Newtonian simulations, we are computing forces and using *f*=*ma* to compute accelerations:

$$\mathbf{a}_i = \mathbf{M}^{-1}\mathbf{f}(t_i, y_i)$$

- Where **M** is a diagonal matrix of masses and **f**() is a function that evaluates the current forces in the system
- We then need to integrate twice to get the final position:

$$\mathbf{v}_{i+1} = \mathbf{v}_i + \mathbf{a}_i \Delta t$$
$$\mathbf{r}_{i+1} = \mathbf{r}_i + \mathbf{v}_{i+1} \Delta t$$

Midpoint Method

- The *explicit midpoint method* first steps halfway using the forward Euler method:

$$y_{i+\frac{1}{2}} = y_i + f(t_i, y_i)\frac{\Delta t}{2}$$

- It then evaluates the derivative at this time, halfway through the full step
- Finally, it re-integrates from the start using the midway derivative

$$y_{i+1} = y_i + f\left(t_{i+\frac{1}{2}}, y_{i+\frac{1}{2}}\right)\Delta t$$

- The idea is that it should improve over the basic Euler method which only looks at the derivative at the beginning of the time step. In the midpoint method, it estimates the derivative in the middle of the time step and uses that instead.
- It requires two force evaluations per time step, whereas the basic Euler method requires only one

Heun’s Method

- Huen’s method is similar to the midpoint method and is also referred to as the explicit trapezoid method
- It starts with a full forward Euler step, and then evaluates the accelerations at the end of the time step
- It then re-integrates using the average of the initial and final accelerations

$$\tilde{y}_{i+1} = y_i + f(t_i, y_i)\Delta t$$

$$y_{i+1} = y_i + \frac{1}{2}[f(t_i, y_i) + f(t_{i+1}, \tilde{y}_{i+1})]\Delta t$$

Runge-Kutta Methods

- Runge-Kutta methods are a family of integrators that combine multiple derivative estimates within the time step
- They can be constructed according to a set of basic patterns, and they generalize a number of other integration schemes
- A popular method is the fourth-order *classical Runge-Kutta method* also known as *RK4*
- RK4 computes accelerations at 4 different points and combines them into a final acceleration

RK4

- A widely used Runge-Kutta method is the classic fourth order RK4 method
- This involves computing 4 different derivatives within the time step and combining them to get a final result

$$k_1 = f(t_i, y_i)\Delta t$$

$$k_2 = f\left(t_{i+\frac{1}{2}}, y_i + \frac{k_1}{2}\right)\Delta t$$

$$k_3 = f\left(t_{i+\frac{1}{2}}, y_i + \frac{k_2}{2}\right)\Delta t$$

$$k_4 = f(t_{i+1}, y_i + k_3)\Delta t$$

$$y_{i+1} = y_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

Verlet Integration

- *Verlet integration* is a technique used specifically for integrating Newton's equations of motion
- It is often used in simulations with many particles, such as galaxies and molecular simulations

$$x(t_{i+1}) = x(t_i) + v(t_i)\Delta t + \frac{1}{2}a(t_i)\Delta t^2$$

$$v(t_{i+1}) = v(t_i) + \frac{a(t_i) + a(t_{i+1})}{2}\Delta t$$

- The method is not well suited to situations where the acceleration is dependent on the velocity (as with dampers), and so has limited application (but can still be used on galaxies & molecules as they don't involve damping)
- It is also very similar to a method called *Leapfrog integration* that evaluates the position at each time step, but evaluates the velocity at the halfway point between time steps
- There are also higher order (more accurate) extensions known as *Yoshida integrators*

Explicit Methods

- All of the methods we have looked at so far are classified as *explicit* methods
- Explicit methods calculate the state at a later time by using information from the current time
- With these methods, we can calculate the derivatives directly (explicitly)
- Even the Runge-Kutta method which involves computing several derivatives at different times is an explicit method, because we are always evaluating the derivatives explicitly and extrapolating forward from there

Implicit Methods

- In contrast to explicit methods, *implicit* methods require solving a (often non-linear) equation involving both the current and future states
- Implicit methods require additional computation over explicit methods and they are typically a lot harder to implement
- Their strength, however, is that they can typically handle much stiffer equations than explicit methods and thus can use much larger time steps
- They may take a lot longer per time step, but they can require much fewer time steps, resulting in a net gain in certain circumstances

Backward Euler Integration

- Recall the forward (or explicit) Euler method:

$$y_{i+1} = y_i + f(t_i, y_i)\Delta t$$

- At first glance, the backward (or implicit) Euler method looks similar to the forward Euler method:

$$y_{i+1} = y_i + f(t_{i+1}, y_{i+1})\Delta t$$

- It involves using a single value of the derivative to advance the entire time step
- However, it requires knowing (solving) the derivative at the end of the time step
- In other words, we must find a new value of y_{i+1} such that the derivative at that time points backwards to the current value of y_i
- This isn't so bad for single scalar equations the way we wrote it above, but for large systems, we need all of the derivatives in the entire configuration at time t_{i+1} to point back to the configuration at time t_i
- This involves solving a (potentially) large system of (probably) nonlinear equations

Backward Euler Method

$$y_{i+1} = y_i + f(t_{i+1}, y_{i+1})\Delta t$$

- The backward Euler method is an example of an *implicit* integration method
- Unlike explicit methods which simply evaluate derivatives at the current state and use that to advance forward, implicit methods *solve* the future state
- This usually involves iterative techniques such as Newton’s method
- As mentioned, these methods are more complex, but they can be very effective in dealing with stiff situations that are difficult to handle with other methods
- We will come back to these in a later lecture

Linear Multistep Methods

- *Linear multistep methods* are a category of integration techniques that keep track of previous derivatives in order to achieve better accuracy with little additional cost
- They generally involve computing one new derivative per time step, but they can use the previous derivatives to fit a smooth curve through the points instead of a straight line approximation
- As they essentially fit a curve through several points, these methods require the function to be smooth and so they don't work well in situations with discontinuities (such as collisions)
- These methods were largely developed by British mathematician John Couch Adams in the 19th century (he's also known for predicting the existence and position of Neptune)

Two-Step Adams-Bashforth

- A simple multistep method is the two-step Adams-Bashforth method:

$$y_{i+2} = y_{i+1} + \left[\frac{3}{2} f(t_{i+1}, y_{i+1}) - \frac{1}{2} f(t_i, y_i)\right] \Delta t$$

- It requires keeping track of the previous value of $f(t_i, y_i)$ so there is some additional complexity involved with using it
- For example, as y_0 is given as the initial value, we can't use this method to compute y_1 so we have to start off with one step of something else (such as forward Euler) and use this method for y_2, and so on
- This can also be an issue when dealing with sharp discontinuities in the simulation that might result from things like collisions or fractures

Higher Order Adams-Bashforth

- The Adams-Bashforth methods are a whole family of integration techniques that extend this further
- Like the two-step method, they require storing previous derivatives, but they have the advantage of only requiring one new derivative evaluation per time step

$$y_{i+1} = y_i + f(t_i, y_i)\Delta t$$

$$y_{i+2} = y_{i+1} + \left[\frac{3}{2} f(t_{i+1}, y_{i+1}) - \frac{1}{2} f(t_i, y_i)\right]\Delta t$$

$$y_{i+3} = y_{i+2} + \left[\frac{23}{12} f(t_{i+2}, y_{i+2}) - \frac{16}{12} f(t_{i+1}, y_{i+1}) + \frac{5}{12} f(t_i, y_i)\right]\Delta t$$

$$y_{i+4} = y_{i+3} + \left[\frac{55}{24} f(t_{i+3}, y_{i+3}) - \frac{59}{24} f(t_{i+2}, y_{i+2}) + \frac{37}{24} f(t_{i+1}, y_{i+1}) - \frac{9}{24} f(t_i, y_i)\right]\Delta t$$

Adams-Moulton

- The Adams-Bashforth methods are examples of explicit methods, as they evaluate derivatives based on known configurations
- The Adams-Moulton methods are multipoint methods like Adams-Bashforth, but they solve the new derivatives implicitly

General Linear Methods

- Around 1965, John Butcher developed a generalization of integration techniques
- He came up with a technique for representing all of the integration methods we've discussed so far in to a single category called *general linear methods*
- GLMs are a superset that includes Runge-Kutta methods, multipoint (Adams) methods, as well as other explicit and implicit methods
- This provides a consistent framework for analyzing and comparing different techniques
- It also provides a process for developing new methods

Discontinuities

- When performing numerical integration, we prefer to work with nice smooth functions that allow us to compute accurate results.
- With smooth functions, we can compute derivatives locally and be reasonably sure that they describe the local shape of the function.
- However, some simulations include discontinuities that break the assumption of smoothness.
- A common example in an impact collision that may cause a sudden change in velocity
- Discontinuities can be problems for any method that requires combining multiple derivatives (such as Adams and Runge-Kutta methods)
- One approach is to use these high quality methods when possible, but switch to a more basic method (such as forward Euler) when discontinuities occur. This can be done on a per-frame, per-DOF basis, and so it will lead to more complicated implementations

Adaptive Time Steps

Adaptive Time Steps

- So far, we’ve assumed that the time step is just set to some constant value, selected as to balance accuracy with computation time.
- This works fine for a lot of situations, as long as the time step is small enough to produce stable, accurate (enough) results.
- However, there is no strict requirement that the time step be constant, and we can often gain an advantage by adapting the time step automatically as the simulation runs.

Adaptive Time Steps

- There are some general guidelines for developing adaptive time stepping schemes, but often they are coupled to specific integrators. For example, there are adaptive versions of Runge-Kutta methods
- Adaptive time step methods typically require some way to estimate the error to make the determination of whether or not to change the time step
- Error estimation is often based on the Taylor series expansion of the integration scheme
- Forward Euler, for example, uses a straight line approximation to the function, which is a first order approximation (essentially, a Taylor series where the higher order terms get truncated)
- The error in this case is going to be on the order of Δt^2
- We can use this to derive a estimation of the error at any given point in the simulation

Adaptive Time Steps

- Typical adaptive schemes keep track of the time step size from frame to frame
- They also have a threshold value limiting the maximum local truncation error

- First, the simulation is advanced using the current time step value
- Then, the error is calculated
- If the error is significantly above the threshold (like 2x or more), then the step is thrown away, the step size is cut in half, and then the frame is recomputed with the new time step
- If the error is slightly above the threshold (like 1x-2x), then the frame is kept, but the time step is cut in half for the next frame
- If the error is well below the threshold (like <1/2), then the time step is doubled for the next frame
- Otherwise, the time step stays the same for the next frame
- This continues on for each frame of the simulation, adapting the time step as necessary
- Of course, you will probably want some upper/lower bounds on this to prevent the time step from getting too big or too small

Simulation Architecture

Architecture Goals

- We’ve seen that there are a lot of different approaches to integration, and that it’s not always clear which method is best in a particular situation.
- Therefore, it would be nice if a simulation library offered the ability to choose between different integration techniques.
- We also want to simulate different things such as solids, fluids, rigid bodies, that each have their own equations of motions.
- Therefore, we need to think about ways that we can architect the software to allow us to combine integration techniques with different equations of motion.
- We should also consider that elaborate physics simulations can be very costly in terms of computation time, so we should also take performance into consideration when discussing architecture.

Integrators & Physical Systems

- It is common in various physics engines to separate out an integrator class that can implement different schemes
- If done properly, a single integrator class can be shared by particle, soft body, rigid body, and fluid simulations, as well as simulations that combine multiple types
- To integrate a physical system in this way, we must group all of its degrees of freedom into vectors that can be manipulated by the integrator
- If a system has *N* degrees of freedom, then it needs to provide an *N* dimensional position and velocity vector to the integrator
- It will also need functions for setting those values from a vector
- In addition, it will need a function that computes all accelerations in the system, based on the current position & velocity values

Physics System Class

```
// Base system
class PhysicsSystem {
public:
        virtual int GetNumDofs();
        virtual void GetPositions(std::vector<float> &pos);
        virtual void GetVelocities(std::vector<float> &vel);

        virtual void SetPositions(const std::vector<float> &pos);
        virtual void SetVelocities(const std::vector<float> &vel);

        virtual void ComputeAccelerations(std::vector<float> &acc);
};

// Derived systems
class MassSpringSystem : public PhysicsSystem{…};
class RigidBodySystem : public PhysicsSystem {…};
```

Integrator Class

```
// Base integrator
class Integrator {
public:
		virtual void Integrate(PhysicsSystem &system, float timestep);
};

// Derived integrators
class ForwardEulerIntegrator : public Integrator {…};
class RungeKuttaIntegrator : public Integrator {…};
class MidpointIntegrator : public Integrator {…};
```

Forward Euler Example

```
Void ForwardEulerIntegrator::Integrate(PhysicsSystem &system, float timestep) {
        // Get positions & velocities
        int numDofs = system.GetNumDofs();
        std::vector<float> pos(numDofs) , vel(numDofs);
        system.GetPositions(pos);
        system.GetVelocities(vel);

        // Compute accelerations
        std::vector<float> acc(numDofs);
        system.ComputeAccelerations(acc);

        // Forward Euler step
        for(int i=0; i<numDofs; i++) {
                vel[i] += acc[i] * timestep;
                pos[i] += vel[i] * timestep;
        }

        // Store results
        system.SetPositions(pos);
        system.SetVelocities(vel);
}
```

Integrator Notes

- Separating out the integrator from the physics system is pretty common practice in general purpose physics engines
- It’s a nice way to do things, but can become quite complicated in some cases:
  - Discontinuities (arising from collisions, etc.)
  - Changing number of DOFs (as in fracture simulations, etc.)
  - Coupled simulations involving different dynamics. For example, if you combine rigid bodies and fluid dynamics, you might want to use different integrators tuned to each case
  - Implicit integrators can benefit from tighter coupling with the physics equations to optimize the solver

Resources

- "Numerical Methods for Ordinary Differential Equations – Third Edition", by John Butcher, 2016
- This is a very serious book by one of the leading experts in the field

- Wikipedia has a lot of good pages on integrators. The page on 'Euler method' is a good place to start and has a good overview of integration, step sizes, approximation error, stability, and accuracy issues